CS3025 Compiladores

Semana 11:
Generacion de Codigo:
The Basics
24 Octubre 2023

Igor Siveroni

Esta clase:

Recordemos:

- El lenguaje objecto: SVML, sintaxis y semántica.
- El lenguaje fuente: IMP0, sintaxis concreta y abstracta

Generacion de código fuente:

- Definicion de `codegen` e implementación.
- Notas sobre correctitud.

El lenguaje objeto: SVML, lenguaje para una maquina de pila virtual

Empecemos definiendo nuestro lenguaje objeto, SVML.
SVML, el lenguaje de la maquina de pila virtual SML, esta definido por la siguiente sintaxis:

```
PSVM ::= Instruction+
Instruction ::= Label? (<instr0> | <instr1>  <num> | <instr2> <id>) <eol>
<instr0> ::= skip | pop | dup | swap | add | sub | mul | div
             gt | ge | lt | le | eq | print
<instr1> ::= push | store | load
<instr2> ::= jmpz | jmpn| goto
```

Un programa SVML esta compuesto por una secuencia de instrucciones separadas por cambio de línea.
La sintaxis de arriba divide a las instrucciones en 3 tipos, dependiendo del numero (0 o 1) de argumentos y tipo de argumento (num o id) que aceptan.
Además, cada instrucción puede estar anotada por un Label (<id>).

Este es un subconjunto de SVML – será suficiente por ahora.

SVML: Semantica – La pila de datos

La característica principal de la SVM es el uso de una pila de operandos: todas las operaciones de la SVM toman sus argumentos de la pila de datos S (stack).

Para mas claridad, será conveniente expresar la pila como `S = d1:…:dn`, donde `dn` corresponde al ultimo elemento de la pila.
También será útil escribir `S:top` y `S:next:top` donde, en ambos casos, `top` es el ultimo elemento de la pila.
Las operaciones que manipulan directamente la pila son:

- `push n`: coloca al numero n arriba de la pila.
  `S` cambia a `S:n`
- `pop`: remueve el ultimo elemento de la pila.
  `S:top` cambia a `S`
- `dup`: duplica el ultimo elemento de la pila.
  `S:n` cambia a `S:n:n`
- `swap`: intercambia los 2 últimos elementos de la pila.
  `S:next:top` cambia a `S:top:next`

SVML: Semantica – Operaciones binarias

Las operaciones aritméticas y de comparación toman, y remueven, 2 argumentos de la pila, y dejan el resultado encima de la pila. Dada la operación `op`, podemos expresar la ejecución de la operación de la siguiente manera:

`S:next:top` cambia a `S:(next |op| top)`

Donde op = `{ add, sub, mul, div, lt, le, gt, ge, eq }`, y

- |op| representa la operación aritmética o comparación eg. + o <.
- Las comparaciones evalúan a 0 o 1.
- Es importante notar el orden de los operandos. Por ejemplo
  `push 4; push 2; sub;` deja `(4-2)` arriba de la pila, y
  `push 5; push 10; lt;` deja el resultado de `(5<10)` arriba de la pila, es decir, 1.

SVML: Semantica – Cambios en el flujo de control

La ejecución de un programa en SVM, por defecto, es secuencial. Si consideramos a un programa de la SVM como una secuencia de instrucciones P indexadas por el registro PC (program counter), entonces el loop de ejecución de la maquina virtual puede escribirse así:

```
while(PC < |P|) { I=P[PC]; PC++; execute(I); }
```

Existen 3 instrucciones que modifican el flujo normal del programa.

- `goto L`: Cambia el flujo del programa a la instrucción anotada con (el label) `L`.
- `jmpz L` : Cambia el flujo del programa a la instrucción anotada con (el label) `L`, si el elemento `top` de la pila es 0. Remueve el ultimo elemento de la pila.
- `jmpn L` : Cambia el flujo del programa a la instrucción anotada con (el label) `L`, si el elemento `top` de la pila NO es 0. Remueve el ultimo elemento de la pila.

En los 3 casos, el valor a PC cambia a `idx(L)`, la posición de la instrucción anotada por `L`.
A `jmpz` y `jmpn` los llamamos saltos condicionales.

SVML: Semantica – Acceso a Memoria

Además de la pila de datos, la SVM posea una sección de memoria que puede ser leída y escrita por medio de un índice: la dirección de memoria. Representemos, por ahora, a la memoria como un arreglo `M`. Dada una dirección de memoria `a`, podemos acceder al valor guardado en la dirección `a` escribiendo `M[a]`. Del mismo modo podemos guardar un valor en la posición `a` escribiendo `M[a] := v`.

Las instrucciones que acceden a la memoria M de la SVM son:

- `load a`: Lee el valor guardado en memoria en la dirección `a` y lo coloca encima de la pila.

  `S` cambia a `S:M[a]`

- `store a`: Escribe en memoria, en la dirección a, el valor `top` de la pila. Remueve `top` de la pila.

  `S:top` cambia a `S` `M[a] = top`

IMP0, nuestro lenguaje fuente

Hemos estado trabajado con el lenguaje IMP, un lenguaje con declaraciones de funciones globales.
Para esta primera parte de la fase de generación de código, trabajaremos con una versión simplificada de IMP, IMP0:

```
Program ::= Body
Body ::= VarDecList StmList
VarDecList ::= (VarDec)*
VarDec ::= “var” Type VarList “;”
Type ::= bool | int
StmList ::= Stm (“;” Stm)*
```

Donde un programa es un bloque `Body`, definido como una lista de declaraciones de variables (globales para el primer bloque) seguidas de una secuencia no-vacia de sentencias.

IMP0 será nuestro lenguaje fuente.

El lenguaje IMP: sentencias y expresiones

La gramática de sentencias es:

```
Stm ::=  AssignStm | PrintStm | IfStm | WhileStm | ReturnStm
AssignStm ::= id “=” Exp
PrintStm ::= “print” “(“ Exp “)”
IfStm ::= “if” Exp “then” Body [“else” Body] “endif”
WhileStm ::= “while” Exp “do” Body “endwhile”
```

La gramática de expresiones, con cuestiones de asociatividad y orden de precedencia resueltas, es:

```
Exp ::=  NumberExp | BoolExp | IdExp | CondExp | ParenthExp | BinExp
NumberExp ::= num     BoolExp ::= “true” | “false”     IdExp ::= id
BinExp ::= Exp op Exp, op in {plus, minus, mul, div, lt, leq, eq }
CondExp ::= “ifexp” “(“ Exp “,” Exp “,” Exp “)”
ParenthExp ::= “(“ Exp “)”
```

IMP0: Generacion de Codigo

Para especificar el generación de código de programas IMP a SVML, definiremos la función `codegen` de la siguiente manera:

```
codegen: Aenv x Exp -> Instruction+
codegen: Aenv x Stm -> Instruction+
codegen: Aenv x Body -> Instruction+
codegen: Aenv x Program -> Instruction+
```

Es decir, `codegen` mapea elementos de IMP a secuencias de instrucciones de SVML.
Para esta transformación es necesario tener un environment de direcciones de memoria Aenv

```
addr in Aenv : id -> int
```

Que mantendrá un registro de las posiciones en memoria de todas las variables vivas, en cualquier punto durante la generación de código del programa.

Además, será conveniente tener una función getLabel()->Label que generara “fresh labels”.

IMP-SVML: Generacion de codigo SVML para expresiones. Sintaxis abstracta

Empezaremos con la definición de `codegen` para expresiones.
Para facilitar la especificación de las reglas de generación de código, expresaremos la sintaxis de IMP usando un tipo especial de sintaxis abstracta (abstract syntax).

```
e in Exp ::=
      | NumberExp(n), n in Int
      | BoolExp(b)  , b in { T, F }
      | IdExp(id)   , id in ID
      | CondExp(e0, e1, e2) |
      | BinExp(e1, e2, op) ,
                op in {plus, minus, mul, div, lt, leq, eq }
      | ParenthExp(e)
```

Generacion de codigo SVML para expresiones.

A continuación definiremos `codegen` para expresiones.

```
codegen: Aenv x Exp -> Instruction+
cg in Instruction+
```

Un punto importante a considerar al definir `codegen` es tener cierta noción de correctitud en mente. En el caso de expresiones, sabemos que toda expresión evalúa a un valor (entero o boolean). Escribamos:

$$eval(env,e) = n$$

Para expresar el hecho que la expresión e evalua a n bajo el environment env (variables a valores). Del mismo modo, dado un código cg del SVML, podemos expresar la ejecución de dicho código de la siguiente manera:

$$\langle S, cg \rangle \rightarrow^* S'$$

Es decir, especificando los cambios en la pila - en realidad, hay otros elementos en juego pero, por ahora, esto es suficiente para expresar este punto. Queremos:

`cg = codegen(addr, e), n=eval(env,e)` entonces $\langle S, cg \rangle \rightarrow^* S:n$

Generacion de codigo SVML: Constantes

Empecemos con números y constantes booleanas:

```
codegen(addr, NumberExp(n)) = push n

codegen(addr, BoolExp(b)) = push b , b in {0,1}
```

La condición de correctitud se satisface trivialmente dado que la ejecución de push deja encima de la pila a n (o b), precisamente el resultado de interpretar la constante n (o b). Adelantándonos un poco,

```
<S,push n> → S:n ^ eval(env,n) = n
```

La implementación es trivial:

```
int ImpCodeGen::visit(NumberExp* e) {
  codegen(nolabel,"push ",e->value);
  return 0;
}
```

```
int ImpCodeGen::visit(BoolConstExp* e) {
  codegen(nolabel,"push ",e->value?1:0);
  return 0;
}
```

Generacion de codigo SVML: Variables y acceso a memoria

Sabemos que:

- Cada variable tiene reservada un espacio de memoria en la SVM.
- Las posiciones de memoria asignadas a variables son guardadas en el environment `addr`.
- La posición de memoria asignada a la variable `id` es `addr(id)`.
- La memoria de la SVM puede ser leída o modificada mediante **`load`** y **`store`**.

  Asi:

```
codegen(addr, IdExp(d)) = load addr(id)
```

El compilador implementa `addr` mediante la variable `direcciones` definida como

```
Environment<int> direcciones;
```

La compilación de IdExp(id) es también bastante sencilla:

```
int ImpCodeGen::visit(IdExp* e) {
  codegen(nolabel,"load",direcciones.lookup(e->id));
  return 0;
}
```

Generacion de codigo SVML: Expresiones Aritmeticas Binarias

Expresiones aritméticas binarias:

- El código generado por la compilación de `BinExp(e1,e2,op)` debe de acabar con **`op`**, la operación correspondiente en SVML.
- Además, `op` espera dos argumentos en la pila, los valores correspondientes a la evaluación de las expresiones `e1` y `e2` (los argumentos).
- Dadas las propiedads de `codegen(addr,e)`, podemos definir la compilación de `BinExp(e1,e2,op)` de la siguiente manera:

```
codegen(addr, BinExp(e1,e2,op)) = codegen(addr,e2)
                                  codegen(addr,e2)
                                  op
```

Asi:

```
<S,cg> →* <S:n1:n2,op> →* S:op(n1,n2)
       cg = codegen(addr, BinExp(e1,e2,op))
       eval(env,e1) = n1
       eval(env,e2) = n2
```

Generacion de codigo SVML: Expresiones Aritmeticas Binarias

La implementación de la compilación de BinExp(id) es :

```
int ImpCodeGen::visit(BinaryExp* e) {
  e->left->accept(this);
  e->right->accept(this);
  string op = "";
  switch(e->op) {
  case PLUS: op =  "add"; break;
  case MINUS: op = "sub"; break;
  case MULT:  op = "mul"; break;
  case DIV:  op = "div"; break;
  case LT:  op = "lt"; break;
  case LTEQ: op = "le"; break;
  case EQ:  op = "eq"; break;
  default: cout << "binop " << Exp::binopToString(e->op) << " not
implemented" << endl;
  }
  codegen(nolabel, op);
  return 0;
}
```

Generacion de codigo SVML: Expresion condicional y jumps

La expresión condicional `CondExp(e0,e1,e2)` evalua al valor de la evaluación de `e1` o `e2`, dependiendo si el valor de la evaluación de `e0` (un boolean) es 1 o 0, respectivamente.

Definimos la compilación de `CondExp(e0, e1,e2)` de la siguiente manera:

```
codegen(addr, CondExp(e0,e1,e2)) =
        codegen(addr,e0)     // evalua a 0 o 1
        jmpz L2              // saltar a e2 si es falso
        codegen(addr,e1)     // evaluar e1
        goto LEND
        L2: skip
        codegen(addr,e2)
        LEND: skip
```

Donde L2 y LEND son etiquetas nuevas (fresh labels)

Generacion de codigo SVML: Expresion condicional y jumps

Nótese que la siguiente definición `CondExp(e0, e1,e2)` es equivalente:

```
codegen(addr, CondExp(e0,e1,e2)) =
        codegen(addr,e0)
        jmpn L1
        codegen(addr,e2)
        goto LEND
        L2: skip
        codegen(add,e1)
        LEND: skip
```

Pero preferimos la primera.

Generacion de codigo SVML: Expresion condicional y jumps

La implementación de la compilación de CondExp(…) es :

```
int ImpCodeGen::visit(CondExp* e) {
  string l1 = next_label();
  string l2 = next_label();

  e->cond->accept(this);
  codegen(nolabel, "jmpz", l1);

  e->etrue->accept(this);
  codegen(nolabel, "goto", l2);

  codegen(l1,"skip");
  e->efalse->accept(this);
  codegen(l2, "skip");

  return 0;
}
```

Generacion de codigo SVML: Sentencias

Del mismo modo que lo hecho con expresiones, expresaremos la sintaxis de las sentencias usando sintaxis abstracta (abstract syntax):

```
s in Stm ::=
       | AssignStm(id,e)
       | PrintStm(e)
       | IfStm(e, bd1, bd2) , bd1, bd2 in Body
       | WhileStm(e, bd)    , bd in Body
       | ReturnExp(e)

b in Body    ::= Body(vdlist, slist) , vdlist in VardDec*
vd in VarDec ::= (T, id)
```

Generacion de codigo SVML: PrintStm

La sentencia `PrintStm(id,e)` imprime (manda a standard output) el resultado de la evaluación de e. SVML tiene una instrucción análoga, **`print`**, que imprime (y remueve) el ultimo elemento de la pila.

Definimos la compilación de `PrintStm(e)` de la siguiente manera:

```
codegen(addr, PrintStm(e)) =
        codegen(addr,e)
        print
```

Así:

`<S, cg;print> →* <S:n, print> → S , cg = codegen(addr, PrintStm(id,e))`

Esto pone en énfasis una característica del código compilado de sentencias: No modifica la pila.

La implementación es un fiel reflejo de la especificación:

```
int ImpCodeGen::visit(PrintStatement* s) {
  s->e->accept(this);
  code << "print" << endl;;
  return 0;
}
```

Generacion de codigo SVML: AssignStm

La sentencia `AssignStm(id,e)`, sintaxis abstracta de `id = e`, guarda el valor resultado de la evaluación de `e` en la dirección de memoria asociada con `id`.

Definimos la compilación de `AssignStm(id, e)` de la siguiente manera:

```
codegen(addr, AssgnStm(id,e)) =
        codegen(addr,e)      // evalua a valor del rhs e
        store addr(id)       // guarda el valor de e en la dirección de id
```

La implementación utiliza el mapping de identificadores a direcciones de memoria `direcciones`:

```
int ImpCodeGen::visit(AssignStatement* s) {
  s->rhs->accept(this);
  codegen(nolabel,"store",direcciones.lookup(s->id));
  return 0;
}
```

Generacion de codigo SVML: Sentencia IfStm

La sentencia condicional `IfStm(e,bd1,bd2)` ejecuta el bloque `e1` o `e2`, dependiendo si el valor de la evaluación de `e` (un boolean) es 1 o 0, respectivamente.

Definimos la compilación de `IfStm(e, bd1,bd2)` de la siguiente manera:

```
codegen(addr, IfStm(e,bd1,bd2)) =
        codegen(addr,e)      // evalua a 0 o 1
        jmpz L2              // saltar a bd2 si es falso
        codegen(addr,bd1)    // ejecutar bd1
        goto LEND
        L2: skip
        codegen(addr,e2)
        LEND: skip
```

Donde L2 y LEND son etiquetas nuevas (fresh labels)

Generacion de codigo SVML: Expresion condicional y jumps

La implementación de la compilación de IfStm(…) es :

```
int ImpCodeGen::visit(IfStatement* s) {
  string l1 = next_label();
  string l2 = next_label();

  s->cond->accept(this);
  codegen(nolabel,"jmpz",l1);
  s->tbody->accept(this);
  codegen(nolabel,"goto",l2);
  codegen(l1,"skip");
  if (s->fbody!=NULL) {
    s->fbody->accept(this);
  }
  codegen(l2,"skip");

  return 0;
}
```

Generacion de codigo SVML: Sentencia WhileStm

La sentencia condicional `WhileStm(e,bd)` ejecuta el bloque `e1` o `e2`, dependiendo si el valor de la evaluación de `e` (un boolean) es 1 o 0, respectivamente.

Definimos la compilación de `WhileStm(e, bd)` de la siguiente manera:

```
codegen(addr, WhileStm(e,bd)) =
        LENTRY: skip
        codegen(addr,e)      // evalua a 0 o 1
        jmpz LEND            // saltar a LEND si es falso
        codegen(addr,bd)     // ejecutar bd
        goto LENTRY
        LEND: skip
```

Donde L2 y LEND son etiquetas nuevas (fresh labels)

Generacion de codigo SVML: Expresion condicional y jumps

La implementación de la compilación de WhileStm(…) es :

```
int ImpCodeGen::visit(WhileStatement* s) {
  string l1 = next_label();
  string l2 = next_label();

  codegen(l1,"skip");
  s->cond->accept(this);
  codegen(nolabel,"jmpz",l2);
  s->body->accept(this);
  codegen(nolabel,"goto",l1);
  codegen(l2,"skip");

  return 0;
}
```

Generacion de codigo SVML: Body

La ejecución de un `BodyStm(vdlist,slist)` esta compuesta de 2 partes: el procesamiento de las declaraciones vdlist y la ejecución de las sentencias en slist.

Definimos la compilación de `BodyStm(e, bd)` de la siguiente manera:

```
codegen(addr, BodyStm(vdlist,s1:…:sn)) =
      codegen(addr,s1)
      …
      codegen(addr,sn)
```

Donde `addr` tiene asignadas direcciones únicas a las variables definidas en `vdlist`, y al resto de variables que están vivas o en *scope*.
Algunas propiedades:

```
Vx in vdlist, x in Dom(addr)
Vx,y in Dom(addr), addr(x) != addr(y)
```

`addr` puede ser calculado por un análisis separado o por el mismo `codegen` (ver implementación) – hemos obviado esa parte para concentrarnos en la generación de código.

Generacion de codigo SVML: Expresion condicional y jumps

La implementación de la compilación de Body(…) es :

```
int ImpCodeGen::visit(Body * b) {
  b->var_decs->accept(this);
  b->slist->accept(this);
  return 0;
}

int ImpCodeGen::visit(VarDecList* s) {
  list<VarDec*>::iterator it;
  for (it = s->vdlist.begin();
       it != s->vdlist.end(); ++it) {
    (*it)->accept(this);
  }
  return 0;
}
```

```
int ImpCodeGen::visit(VarDec* vd) {
  list<string>::iterator it;
  for (it = vd->vars.begin();
       it != vd->vars.end(); ++it){
    direcciones.add_var(*it,
                        siguiente_direccion++);
  }
  return 0;
}

int ImpCodeGen::visit(StatementList* s) {
  list<Stm*>::iterator it;
  for (it = s->slist.begin();
       it != s->slist.end(); ++it) {
      (*it)->accept(this);
  }
  return 0;
}
```

Notas sobre correctitude (solo notas)

Hemos utilizado ciertas nociones informales de correctitud para expresar algunas propiedades de la generación de código. Por ejemplo, la ejecución de `push` en la SVM la escribimos:

```
<S,push n> → S:n
```

Podríamos también modelar la memoria y agregarla al estado del programa, junto con la pila. Así, la ejecución y compilación de `AssignStm(id,e)` puede explicarse:

```
<S,M, cg;store a> →* <S:n, store a> → <S,M[a->n]>
```

Donde `cg = codegen(addr,e), a=addr(id),n = eval(env,e)`

Esto debe de ir acompañado por alguna relación entre `env` y `M`. Por ejemplo:

```
Vx in env, M[addr(x)] = env(x)
```

También podríamos ampliar la definición del estado de la SVM y describir la ejecución de cada tipo de instrucción I especificando los cambios en el estado total:

```
<P,pc,M,S> → <P,pc',M',S'> , I = P[pc]
```

Donde `P` es el programa SVM y `pc` es el program counter. Estos detalles dependerán de cuan precisos queramos ser en nuestra especificación (y su uso). Lo mismo para `eval(env,e)`.